

---

## Rails Security Checklist

This checklist is limited to Rails security precautions and there are many other aspects of running a Rails app that need to be secured (e.g. up-to-date operating system and other software) that this does not cover. **Consult a security expert.**

One aim for this document is to turn it into a community resource much like the Ruby Style Guide.

**BEWARE** this checklist is not comprehensive and was originally drafted by a Rails developer with an interest in security - not a security expert - so it may have some problems - you have been warned!

### The Checklist (*in no particular order*)

#### Controllers

- ☐ Enable secure default callbacks for `ApplicationController` (and other abstract controllers)
  - ☐ Enforce authentication callbacks on actions (Devise's `authenticate_user!`)
  - ☐ Enforce authorization callbacks on actions (Pundit's `verify_authorized`)
  - ☐ Enforce authorization-related scoping callbacks on actions (Pundit's `verify_policy_scoped`)
  - ☐ Enforce CSRF protections (protect from forgery)
- ☐ When disabling security-related controller callbacks, target actions on a case-by-case basis. Be very selective and deliberate and only disable it in that concrete controller. Avoid sweeping changes in the controller class hierarchy that would make subclasses less secure by default.

#### Routes

- ☐ Perform authentication and authorization checks in `routes.rb`. It is intentional this duplicates many of the security checks you already perform in the controller callbacks (Devise's `authenticate` and `authenticated`) (motivations: defence-in-depth, swiss cheese model).
- ☐ Check all URL endpoints of engines and other Rack apps mounted in `routes.rb` are protected with correct authentication and authorization checks. For sensitive engines/Rack apps favor not leaking they are installed at all by responding with 404 to non-logged in admin users.
- ☐ Check any developer/test-related engines/Rack apps do not expose any URL endpoints in production. They should not even leak (e.g. via 500 HTTP response code) information that they are installed. Ideally don't have their gems installed in production.

---

## Views

- ☐ Avoid HTML comments in view templates as these are viewable to clients. Use server-side comments instead:

```
1 # bad - HTML comments will be visible to users who "View Source":
2 <!-- This will be sent to clients -->
3 <!-- <%= link_to "Admin Site", "https://admin.example.org/login"
   %> -->
4
5 # ok - ERB comments are removed by the server, and so not viewable
   to clients:
6 <%# This will _not_ be sent to clients %>
7 <%#= link_to "Admin Site", "https://admin.example.org/login" %>
```

## URL Secret Tokens

- ☐ Mitigate `Referer` header leaking URL secret tokens to 3rd parties (e.g. password reset URLs can be leaked to CDNs, JS hosted by third parties, other sites you link to) <https://robots.thoughtbot.com/is-your-site-leaking-password-reset-links>

## IDs

- ☐ Avoid exposing sequential IDs (98, 99, 100, ...) which can leak information about your app's usage and assist forced browsing attacks. For example, sequential IDs are often exposed in URLs, form field HTML source, and APIs. Sequential IDs reveal the size and rate at which certain types of data are created in your app. For example, if a competitor signs up to your service and their account page is at path `/users/12000`, and they sign up again in a month, and their new account path is `/users/13000`, you have leaked that your service gains roughly 1,000 sign-ups per month and has 13,000 accounts total. It is not recommended but some small mitigation can be made by starting IDs at a very large number, however this still leaks the rate of new data creation.
- ☐ If IDs need to be exposed in URLs, forms, etc., favor less predictable IDs such as UUIDs or hashes instead of sequential IDs. For files consider using a technique like Paperclip's URI Obfuscation to produce unpredictable file paths (URI Obfuscation will need to be used alongside other protections).
- ☐ Configure Rails model generators to use UUID primary keys by default:

```
1 # In config/application.rb
2 config.generators do |g|
```

---

```
3   g.orm :active_record, primary_key_type: :uuid
4 end
```

## Random Token Generation

- ☐ **CONTRIBUTOR NEEDED** Use [SecureRandom](#) or should we favor <https://github.com/cryptosphere/sysrandom>?

## Logging

- ☐ Avoid Rails insecure default where it operates a blocklist and logs most request parameters. A safelist would be preferable. Set up the `filter_parameters` config to log no request parameters:

```
1 # File: config/initializers/filter_parameter_logging.rb
2 Rails.application.config.filter_parameters += [:password]
3 if Rails.env.production?
4   MATCH_ALL_PARAMS_PATTERN = /.+/
5   Rails.application.config.filter_parameters += [
6     MATCH_ALL_PARAMS_PATTERN
7   ]
8 end
```

- ☐ Regularly audit what data is captured by log files, 3rd party logging, error catching and monitoring services. You (and your users!) may be surprised at what sensitive information you find. Data stored in log files and 3rd party services can be exploited.
- ☐ Favor minimal logging.
- ☐ Consider not archiving logs or regularly purging archived logs stored by you and 3rd parties.

## Input Sanitization

- ☐ Filter and validate all user input
- ☐ Avoid code that reads from filesystem using user-submitted file names and paths. Use a strict safelist of permitted file names and paths if this cannot be avoided.
- ☐ Any routes that redirect to a URL provided in a query string or POST param should operate a safelist of acceptable redirect URLs and/or limit to only redirecting to paths within the app's URL. Do not redirect to any given URL.
- ☐ Consider adding a defensive layer to strong parameters to reject values that do not meet type requirements ([https://github.com/zendesk/stronger\\_parameters](https://github.com/zendesk/stronger_parameters))

- 
- ☐ Consider sanitizing all ActiveRecord attributes (favoring the secure default of an opt-out sanitizer such as `Loofah::XssFoliate` <https://github.com/flavorjones/loofah-activerecord>)

## Markdown Rendering

- ☐ Favor markdown rendering that operates using a safelist of permitted features and forbids rendering arbitrary HTML, especially if you accept markdown input from users.
- ☐ If using RedCarpet, favor `Redcarpet::Render::Safe` over other renderers such as `RedCarpet::Render::HTML`

```
1 # bad
2 renderer = Redcarpet::Render::HTML.new
3
4 # less risky
5 renderer = Redcarpet::Render::Safe.new
```

## Uploads and File Processing

- ☐ Avoid handling file uploads on your (application) servers.
- ☐ Favor scanning uploaded files for viruses/malware using a 3rd party service. don't do this on your own servers.
- ☐ Operate a safelist of allowed file uploads
- ☐ Avoid running imagemagick and other image processing software on your own infrastructure.

## Email

- ☐ Throttle the amount of emails that can be sent to a single user (e.g. some apps allow multiple password reset emails to be sent without restriction to the same user)
- ☐ Avoid user-provided data being sent in emails that could be used in an attack. E.g. URLs will become links in most email clients, so if an attacker enters a URL (even into a field that is not intended to be a URL) and your app sends this to another user, that user/victim may click on the attacker-provided URL.
- ☐ Email security (needs more info)
  - ☐ Use DKIM (<https://scotthelme.co.uk/email-security-dkim/>)
  - ☐ etc.

---

## Detecting Abuse and Fraud

- ☐ Notify users via email when their passwords change | HOWTOs: Devise
- ☐ Favor sending notifications to user for significant account-related events (e.g. password change, credit card change, customer/technical support phone call made, new payment charge, new email or other contact information added, wrong password entered, 2FA disabled/enabled, other settings changes, login from a never-before used region and/or IP address)
- ☐ Do not send the new password via unencrypted email.
- ☐ Consider keeping an audit trail of all significant account-related events (e.g. logins, password changes, etc.) that the user can review (and consider sending this as a monthly summary to them)
- ☐ Use this audit trail or counters to rate-limit dangerous actions. For instance, prevent brute force password attacks by only allowing some maximum number of logins per second.
- ☐ Limit the creation of valuable data, globally, per user, per IP address, per country (IP geolocation), per zip code, per phone number, per social security number, or a combination thereof, to mitigate other kinds of attacks (DDOS, overflow, fraud). For example, only allowing 3 different social security numbers per IP address could reduce fraudulent credit card applications.

## Logins, Registrations

- ☐ Favor multi-factor authentication
- ☐ Favor Yubikey or similar
- ☐ Nudge users towards using multi-factor authentication. Enable as default and/or provide incentives. For example MailChimp give a 10% discount for enabling 2FA.
- ☐ Favor limiting access per IP-address, per device, especially for administrators
- ☐ Require user confirms account (see Devise's confirmable module)
- ☐ Lock account after X failed password attempts (see Devise's lockable module)
- ☐ Timeout logins (see Devise's timeoutable module)
- ☐ Favor mitigating user enumeration (source)
  - Clearance mitigates user enumeration by default (except on registration) (source)
  - Devise needs to be configured to mitigate user enumeration by configuring `paranoid` mode (except on registration) (source)
  - Both Clearance & Devise do not mitigate user enumeration for registration, as apps often want to report if an email address is already registered.

---

## Passwords

- ☐ Favor stronger password hashing with higher workload (e.g. favor more stretches). At time of implementation, research what the currently recommended password hashing algorithms are.
- ☐ Add calendar reminder to regularly review your current password storage practices and whether to migrate to a different mechanism and/or increase the workload as CPU performance increases over time.
- ☐ Prevent password reuse
- ☐ Enforce strong, long passwords
- ☐ Prevent commonly-used passwords (see Discourse codebase for an example)
- ☐ Proactively notify/prevent/reset users reusing passwords they've had compromised on other services (<https://krebsonsecurity.com/2016/06/password-re-user-get-to-get-busy/> - interestingly Netflix apparently uses Scumblr, an open-sourced Rails app, to help with this: <http://techblog.netflix.com/2014/08/announcing-scumblr-and-sketchy-search.html>)
- ☐ Consider adding a layer of encryption to the stored password hashes (and other hashed secrets) (<https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/>)

## Timing Attacks

- ☐ Favor padding/increasing the time it takes to initially login and to report failed password attempts so as to mitigate timing attacks you may be unaware of and to mitigate brute force and user enumeration attempts. See how PayPal shows the “loading...” screen for a good few seconds when you first login (should this always be a fixed set amount of time e.g. 5 seconds and error asking user to try again if it takes longer?)(please correct me on this or add detail as this is an assumption I'm making about the reasons why PayPal do this).
- ☐ Mitigate timing attacks and length leaks on password and other secret checking code <https://thisdata.com/blog/timing-attacks-against-string-comparison/>
- ☐ Avoid using secret tokens for account lookup (includes API token, password reset token, etc.). Do not query the database using the token, this is vulnerable to timing attacks that can reveal the secret to an attacker. Use an alternative identifier that is not the token for the query (e.g. username, email, `api_locator`).

```
1 # bad - timing attack can reveal actual token
2 user = User.find_by(token: submitted_token)
3 authenticated = !user.nil?
4
5 # less risky
6 # step 1: find user by an identifier that is *not* the API key, e.
   g. username, email, api_locator
```

---

```
7 user = User.find_by(username: submitted_username)
8 # step 2: compare tokens taking care to mitigate timing attacks
  and length leaks.
9 # (NB. favor *not* storing the token in plain text)
10 authenticated = ActiveSupport::SecurityUtils.secure_compare(
11   # using digests mitigates length leaks
12   ::Digest::SHA256.hexdigest(user.token),
13   ::Digest::SHA256.hexdigest(submitted_token)
14 )
```

## Databases

- ☐ Beware any hand-written SQL snippets in the app and review for SQL injection vulnerabilities. Ensure SQL is appropriately sanitized using the ActiveRecord provided methods (e.g. see [sanitize\\_sql\\_array](#)).
- ☐ Web application firewall that can detect, prevent, and alert on known SQL injection attempts.
- ☐ Keep Web application firewall rules up-to-date
- ☐ Minimize database privileges/access on user-serving database connections. Consider user accounts, database system OS user account, isolating data by views: [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection)
- ☐ Minimize the data you store on users (especially PII) and regularly review if you can store less or delete older data. E.g. Does the app need to store a user's birthday in perpetuity (or at all) or only need it at registration to check they're old enough? Once data is no longer needed favor deleting it.
- ☐ Remove database links between user profiles and their data if possible: <http://andre.arko.net/2014/09/20/how-to-safely-store-user-data/>
- ☐ Favor storing data encrypted (<https://github.com/rocketjob/symmetric-encryption>)
- ☐ Do not store API keys, tokens, secret questions/answers and other secrets in plain text. Protect via hashing and/or encryption.
- ☐ Encrypted, frequent database backups that are regularly tested can be restored. Consider keeping offline backups.

## Redis

- ☐ Heroku Redis sounds like its insecure by default, secure it as described here using stunnel: <https://devcenter.heroku.com/articles/securing-heroku-redis> (*Does this affect most Redis setups? Are there any Redis providers who are more secure by default?*)
  - Seeking contributors to help with securing Redis, please open a PR and share your experience.

---

## Gems

- ☐ Minimize production dependencies in Gemfile. Move all non-essential production gems into their own groups (usually test and development groups)
- ☐ Run Bundler Audit regularly (and/or Snyk/Dependabot services)
- ☐ Update Bundler Audit vulnerability database regularly
- ☐ Review `bundle outdated` results regularly and act as needed

## Detecting Vulnerabilities

- ☐ Join, act on, and read code for alerts sent by Rails Security mailing list
- ☐ Brakeman (preferably Brakeman Pro) runs and results are reviewed regularly
- ☐ Update Brakeman regularly
- ☐ Security scanning service (e.g. Detectify)
- ☐ Provide ways for security researchers to work with you and report vulnerabilities responsibly

## Software Updates

- ☐ Update to always be on maintained versions of Rails. Many older Rails versions no longer receive security updates.
- ☐ Keep Ruby version up-to-date

## Test Coverage for Security Concerns

- ☐ Test coverage (“AbUser stories”) for security-related code. Including but not limited to:
  - ☐ Account locking after X password attempt failures
  - ☐ Password change notifications sent
  - ☐ URL tampering (e.g. changing id values in the URL)
  - ☐ Attackers, including logged-in users, are blocked from privileged actions and requests. For example, assume a logged-in user who is also an attacker does not need a “Delete” button to submit an HTTP request that would do the same. Attacker-crafted HTTP requests can be mimicked in request specs.

## Cross Site Scripting

- ☐ Regularly grep codebase for `html_safe`, `raw`, etc. usage and review



---

## Developer Hardware

- ☐ Prevent team members from storing production data and secrets on their machines
- ☐ Enable hard disk encryption on team members hardware

## Public, non-production Environments (Staging, Demo, etc.)

- ☐ Secure staging and test environments.
  - ☐ Should not leak data. Favor not using real data in these environments. Favor scrubbing data imported from production.
  - ☐ Avoid reusing secrets that are used in the production environment.
  - ☐ Favor limiting access to staging/test environments to certain IPs and/or other extra protections (e.g. HTTP basic credentials).
  - ☐ Prevent attackers making a genuine purchase on your staging site using well-known test payment methods (e.g. Stripe test credit card numbers)

## Regular Expressions

- ☐ Favor using `\A` and `\Z` as regular expression anchors instead of `^` and `$` (<http://guides.rubyonrails.org/security.html#regular-expressions>)

## Handling Secrets

- ☐ Favor changing secrets when team members leave.
- ☐ Do not commit secrets to version control. Preventative measure: <https://github.com/aws-labs/git-secrets>
- ☐ Purge version control history of any previously committed secrets.
- ☐ Consider changing any secrets that were previously committed to version control.

## Cookies

- ☐ Secure cookie flags
- ☐ Restrict cookie access as much as possible

---

## Headers

- ☐ Secure Headers (see gem)
- ☐ Content Security Policy

## Assets

- ☐ Subresource Integrity for your assets and possibly 3rd party assets [https://developer.mozilla.org/en-US/docs/Web/Security/Subresource\\_Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity)

## TLS/SSL

- ☐ Force TLS/SSL on all URLs, including links, assets, images for internal and 3rd party URLs. No mixed protocols.
- ☐ Use SSL labs to check grade
- ☐ HSTS

## Traffic

- ☐ Rack Attack to limit requests and other security concerns
- ☐ Consider DDOS protections e.g. via CloudFlare

## Contacting Users

- ☐ Have rake task or similar ready to go for mass-password reset that will notify users of issue.
- ☐ Consider having multiple ways of contacting user (e.g. multiple emails) and sending important notifications through all of those channels.

## Regular Practices

- ☐ Add reminders in developer calendars to do the regular security tasks (e.g. those elsewhere in this checklist) and for checking if this checklist has changed recently.

## Further Reading

- ☐ Review and act on OWASPs literature on Ruby on Rails [https://cheatsheetseries.owasp.org/cheatsheets/Ruby\\_o](https://cheatsheetseries.owasp.org/cheatsheets/Ruby_o)

- 
- ☐ More covered at <http://guides.rubyonrails.org/security.html>
  - ☐ See <http://cto-security-checklist.sqreen.io/>
  - ☐ *etc.*

## Reminders

- Security concerns trump developer convenience. If having a secure-defaults `ApplicationController` feels like a pain in the neck when writing a public-facing controller that requires no authentication and no authorization checks, you're doing something right.
- Security is a moving target and is never done.
- The DRY principle is sometimes better ignored in security-related code when it prevents defence-in-depth, e.g. having authentication checks in `routes.rb` and controller callbacks is a form of duplication but provides better defence.

## Contributors

Contributions welcome!